**Menu Pseudocode**

// Switch case statement

Switch case 1: For each line (i) in file {

// Create data structure (courseInfo) with attributes: Course number, course name, prerequisite one, prerequisite two

CourseInfo course\_info;

course\_info.courseNumber = file[i][0];

course\_info.courseName = file[i][1];

course\_info.prereqOne = file[i][2];

course\_info.prereqTwo = file[i][3];

}

Switch case 2: For each course in course\_info {

Print course\_info.courseNumber[i];

}

Switch case 3: For each course in course\_info {

Print course\_info.courseName;

If course\_info.prereqOne != null {

Print course\_info.prereqOne;

}

If course\_info.prereqTwo != null {

Print course\_info.prereqTwo;

}

Switch case 4: Exit

Return 0;

}

**Pseudocode Print Out List**

If (node == NULL)

Return;

printInOrder(node->left);

cout << node->course\_info << “ “;

printInorder(node->right);

**Vector Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

A vector has an advantage of adding an index by pushing the size of the array, giving the data structure a dynamic size. However, the way the loops perform the search algorithm cause for a slower predicted run time.

**Hash Table Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **While there are courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 2 |
| **Runtime** | | | O(n) |

Hash tables are able to parse through data quickly by finding matching keys but once every bucket is filled, searching becomes more tedious since each bucket is searched through using a linked list. A hash table has the best case scenario with a run time of O(1) but a worst case scenario of O(n). If the hash table has quite a bit of data, searching for a specific key could cause for a slow run time.

**Binary Search Tree Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **While node is not null** | 1 | n | n |
| **Traverse left subtree** | 1 | Logn | Logn |
| **Traverse right subtree** | 1 | Logn | Logn |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **Total Cost** | | | n + 2Logn |
| **Runtime** | | | O(Logn) |

Binary search trees have a great best-case time complexity of O(logn), which is the case when the tree is perfectly balanced. If the tree is unbalanced, the worst case run time is O(n). There are 3 methods of traversing a tree which are inorder, preorder, and postorder. In an inorder traversal, the order is the left node, then root, then the right node. In preorder, the root is first, then the left node, then the right node. In postorder traversal the left node is first, then the right node, then the root. With the linearithmic run time, no matter how many nodes the tree has, the run time plateaus at the logn.

References

Croy, M. H. (2021, December 15). *How To Calculate Time Complexity With Big O Notation*. Medium. https://medium.com/dataseries/how-to-calculate-time-complexity-with-big-o-notation-9afe33aa4c46

GeeksforGeeks. (2021, March 8). *Time complexities of different data structures*. https://www.geeksforgeeks.org/time-complexities-of-different-data-structures/